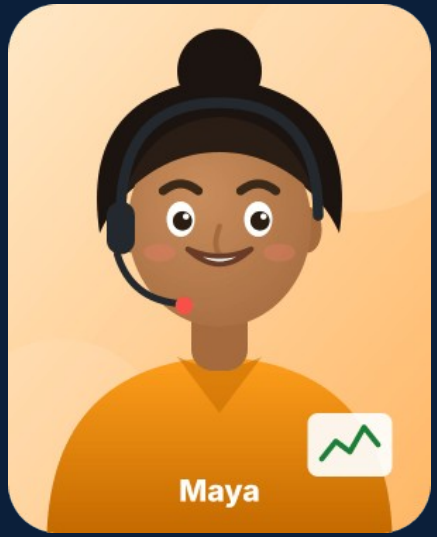
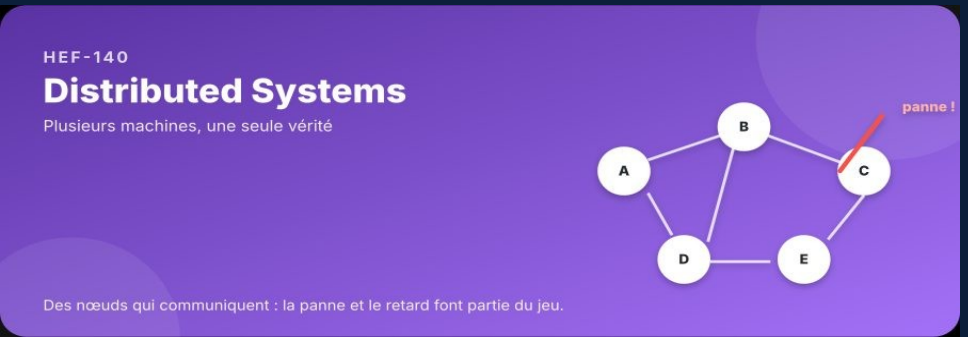
HEF-140 · MISSION 02 / 10

RPC et communication

Appeler une fonction à distance ressemble à un appel local — jusqu'au jour où la réponse ne vient pas. Alors, l'appel a-t-il eu lieu ?





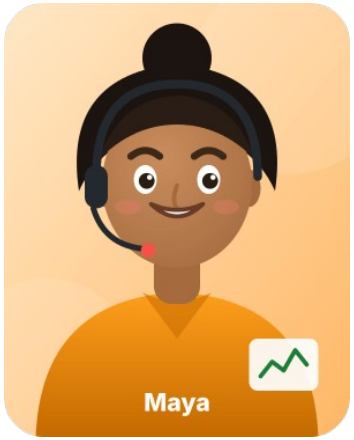
Badge : Messager fiable

100 min · avancé

L'histoire

Le service d'inscription de HashBox appelle le service de paiement : `debiter(membre, 40)`. La réponse n'arrive jamais. Kodi relance l'appel. Trois jours plus tard, le client se plaint d'avoir été débité deux fois : la première requête **avait** réussi, c'est seulement la **réponse** qui s'était perdue.

> *« Un appel à distance a trois issues, dit Sam : il réussit, il échoue… ou on ne sait pas. C'est la troisième qui change toute la conception. Et elle n'existe pas dans un appel local. »*



Tu apprends à construire un mini-RPC avec sockets, à gérer le cadrage des messages, les erreurs et les délais, et à comprendre les trois sémantiques de livraison : au plus une fois, au moins une fois, exactement une fois (et pourquoi elle est une illusion).

Ta mission

À la fin, je serai capable de…

1. comparer les styles de communication : appel synchrone (RPC/REST), messages asynchrones
2. implémenter un RPC avec cadrage par longueur, sérialisation JSON, erreurs et délais d'attente
3. expliquer les sémantiques au plus une fois / au moins une fois / exactement une fois
4. reconnaître les trois issues d'un appel distant (succès, échec, inconnu)
5. faire évoluer un contrat de message de façon compatible

Défi d'échauffement

Défi d'échauffement

Tu envoies une lettre recommandée pour demander un remboursement. Aucun accusé de réception ne revient après trois semaines. Tu hésites : renvoyer la demande (risque d'être remboursé deux fois) ou attendre (risque de ne jamais être remboursé) ? Que ferais-tu pour pouvoir renvoyer sans danger ?

Le concept

Les façons de communiquer

Style	Principe	Avantages	Inconvénients
Appel synchrone (RPC, REST, gRPC)	l'appelant attend la réponse	simple à raisonner, résultat immédiat	couplage temporel, panne en cascade, latences cumulées
Messages asynchrones (files, publication/abonnement)	l'émetteur dépose un message et continue	découplage, tampon face aux pics	plus complexe, cohérence finale
Flux d'événements	un journal ordonné que plusieurs consommateurs lisent	rejouable, découplé	conception d'événements, ordre

Règle utile : **synchrone** quand l'appelant a besoin du résultat pour continuer ; **asynchrone** quand on peut décoller du temps.

Le concept

Un appel à distance n'est pas un appel local

Appel local	Appel distant
Rapide (ns)	lent (ms)
Fiable	peut échouer de plusieurs façons
Mêmes types en mémoire	sérialisation, versions différentes
Résultat certain	succès, échec **ou inconnu**

Les composants d'un RPC

1. **Contrat** (interface) : méthodes, types des paramètres et des résultats (IDL : Protocol Buffers, OpenAPI).
2. **Souche cliente** (*stub*) : transforme l'appel local en message.

Le concept

Les composants d'un RPC (suite)

3. **Sérialisation** : JSON, Protocol Buffers, MessagePack.
4. **Transport** : TCP, HTTP/2.
5. **Squelette serveur** : décode, appelle la vraie fonction, encode la réponse.

Le cadrage des messages sur TCP

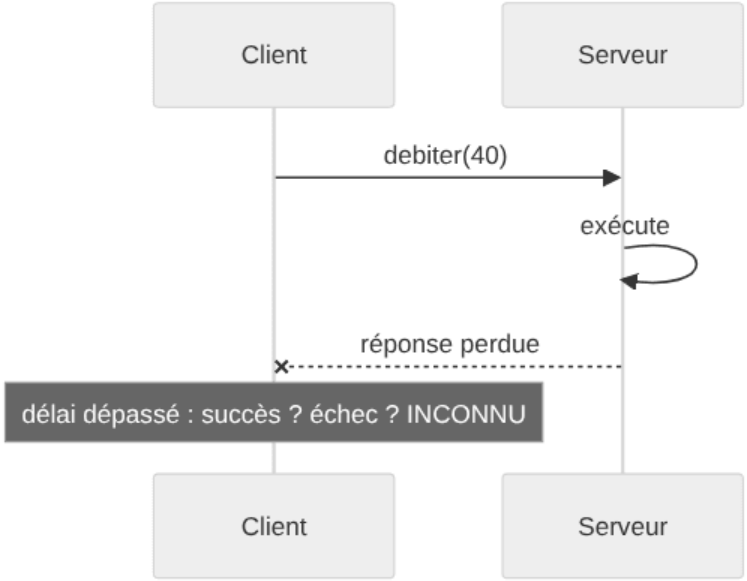
TCP est un **flux d'octets** sans frontières de messages. Il faut donc un **cadrage** :

- **préfixe de longueur** : 4 octets indiquant la taille, puis le contenu ;
- délimiteur (ligne) ;
- HTTP (en-têtes + longueur).

Et lire correctement : `recv(n)` peut renvoyer **moins** de `n` octets ; il faut boucler jusqu'à avoir tout reçu.

Le concept

Les trois issues d'un appel (suite)



Le concept

Les trois issues d'un appel (suite)

Description du schéma : le client envoie une requête ; le serveur l'exécute ; la réponse est perdue ; le client atteint son délai d'attente sans savoir si la requête a été exécutée.

Sémantiques de livraison

Sémantique	Comportement	Comment	Risque
Au plus une fois (*at-most-once*)	exécutée 0 ou 1 fois	pas de relance	action perdue
Au moins une fois (*at-least-once*)	exécutée 1 ou plusieurs fois	relancer jusqu'à réponse	**doublons**
Exactement une fois (*exactly-once*)	exécutée 1 fois	?	impossible à garantir au niveau du transport

Le concept

Sémantiques de livraison (suite)

Exactement une fois n'existe pas comme garantie de réseau. On l'obtient comme **effet** : *au moins une fois* (relances) + *idempotence* (le doublon est sans effet) — mission 4.

Délais et échéances

Un appel doit **toujours** avoir un délai maximal. Mieux : une **échéance** (*deadline*) absolue transmise d'appel en appel (mission 3), pour que chaque maillon sache combien de temps il reste.

Gestion des erreurs

Distinguer :

- **erreurs métier** (« solde insuffisant ») : définitives, ne pas relancer ;
- **erreurs transitoires** (indisponible, délai) : on peut relancer, si l'opération est idempotente ;

Le concept

Gestion des erreurs (suite)

- **erreurs de protocole** (message mal formé).

Évolution des contrats

Les clients et serveurs sont déployés à des moments différents : le contrat doit évoluer de façon **compatible** (cf. HEF-110, mission 5) :

- ajouter un champ **optionnel** : compatible ;
- supprimer ou renommer un champ : cassant ;
- **ignorer les champs inconnus** à la lecture ;
- numéroter les versions des champs (Protocol Buffers).

Le concept

Évolution des contrats (suite)

🧠 ***Astuce*** *: pour chaque appel distant, écris trois lignes : que se passe-t-il si ça échoue ? si ça expire ? si ça s'exécute deux fois ? Si tu ne peux pas répondre aux trois, l'appel n'est pas prêt.*

Exemple pas à pas

Problème : construire un mini-RPC de bout en bout sur sockets locales (cadrage par longueur, JSON, erreurs, délai d'attente), puis rejouer la perte de réponse pour voir l'exécution en double.

```
import json, socket, struct, threading, time

# ---------------- cadrage : 4 octets de longueur + JSON ----------------
def envoyer(sock, objet):
    corps = json.dumps(objet).encode()
    sock.sendall(struct.pack(">I", len(corps)) + corps)

def lire_exactement(sock, n):
    tampon = b""
    while len(tampon) < n:
        morceau = sock.recv(n - len(tampon))                    # recv peut renv
   oyer moins que demandé !
        if not morceau:
            raise ConnectionError("connexion fermée")
        tampon += morceau
```

Exemple pas à pas (suite)

```
    return tampon

def recevoir(sock):
    (taille,) = struct.unpack(">I", lire_exactement(sock, 4))
    return json.loads(lire_exactement(sock, taille))

# ---------------- serveur ----------------
class Serveur:
    def __init__(self, methodes, perdre_premiere_reponse_de=None):
        self.methodes, self.appels, self.perdre = methodes, [], perdre_premiere_re
   ponse_de
        self.deja_perdu = False
        self.sock = socket.socket(); self.sock.bind(("127.0.0.1", 0)); self.sock.l
   isten()
        self.port = self.sock.getsockname()[1]
        threading.Thread(target=self._boucle, daemon=True).start()
    def _boucle(self):
        while True:
            try: conn, _ = self.sock.accept()
```

Exemple pas à pas (suite)

```
            except OSError: return
            threading.Thread(target=self._servir, args=(conn,), daemon=True).start
    ()
     def _servir(self, conn):
         with conn:
             while True:
                 try: requete = recevoir(conn)
                 except (ConnectionError, struct.error, OSError): return
                 try:
                     resultat = self.methodes[requete["methode"]](**requete["params
    "])
                     reponse = {"id": requete["id"], "ok": resultat}
                 except KeyError as e:
                     reponse = {"id": requete["id"], "erreur": {"type": "methode_in
    connue", "message": str(e)}}
                 except Exception as e:
                     reponse = {"id": requete["id"], "erreur": {"type": type(e).__n
    ame__, "message": str(e)}}
                 self.appels.append(requete["methode"])
```

Exemple pas à pas

Exemple pas à pas (suite)

```
                if self.perdre == requete["methode"] and not self.deja_perdu:
                    self.deja_perdu = True; conn.close(); return          # l'acti
   on a eu lieu, mais la réponse est PERDUE
                envoyer(conn, reponse)
    def fermer(self): self.sock.close()

# ---------------- client ----------------
class ErreurDistante(Exception): pass
class DelaiDepasse(Exception): pass

class Client:
    def __init__(self, port, delai=1.0):
        self.port, self.delai, self.compteur = port, delai, 0
    def appeler(self, methode, **params):
        self.compteur += 1
        try:
            with socket.create_connection(("127.0.0.1", self.port), timeout=self.d
   elai) as s:
                s.settimeout(self.delai)
```

Exemple pas à pas (suite)

```
                envoyer(s, {"id": self.compteur, "methode": methode, "params": par
    ams})
                r = recevoir(s)
            except (socket.timeout, TimeoutError):
                raise DelaiDepasse(f"{methode} : pas de réponse en {self.delai} s — ré
    sultat INCONNU") from None
            except ConnectionError:
                raise DelaiDepasse(f"{methode} : connexion perdue — résultat INCONNU")
     from None
            if "erreur" in r:
                raise ErreurDistante(f"{r['erreur']['type']}: {r['erreur']['message']}
    ")
            return r["ok"]
     def __getattr__(self, nom):                                  # souche : clien
    t.addition(a=1, b=2)
            if nom.startswith("_"): raise AttributeError(nom)
            return lambda **params: self.appeler(nom, **params)

SOLDES = {"ada": 100}
```

Exemple pas à pas (suite)

```
def debiter(membre, montant):
    if SOLDES[membre] < montant: raise ValueError("solde insuffisant")
    SOLDES[membre] -= montant; return SOLDES[membre]
def lent(): time.sleep(0.4); return "fini"

srv = Serveur({"addition": lambda a, b: a + b, "debiter": debiter, "lent": lent})
cli = Client(srv.port, delai=0.2)
assert cli.addition(a=2, b=3) == 5                                  # appel réussi
try: cli.debiter(membre="ada", montant=500); raise AssertionError
except ErreurDistante as e: print("erreur métier :", e)               # échec CLAIR
    : solde insuffisant (ne pas relancer)
try: cli.inconnue(); raise AssertionError
except ErreurDistante as e: print("méthode inconnue :", e)
try: cli.lent(); raise AssertionError
except DelaiDepasse as e: print("délai :", e)                         # le serveur tr
   availle, mais le client abandonne : INCONNU
srv.fermer()

# ---------------- la perte de réponse : l'action a lieu DEUX fois ---------------
```

Exemple pas à pas (suite)

```
    -
SOLDES["ada"] = 100
srv = Serveur({"debiter": debiter}, perdre_premiere_reponse_de="debiter")
cli = Client(srv.port, delai=0.5)
try:
    cli.debiter(membre="ada", montant=40)
except DelaiDepasse as e:
    print("1re tentative :", e)
solde = cli.debiter(membre="ada", montant=40)                    # « at-least-on
   ce » : on relance…
print(f"solde final : {solde} (attendu 60 si une seule fois ; on a débité 2 fois)"
   )
assert SOLDES["ada"] == 20 and srv.appels.count("debiter") == 2    # …et l'effet e
   st appliqué deux fois !
srv.fermer()
```

Lecture :

Exemple pas à pas (suite)

1. Le **cadrage** par longueur et la boucle `lire_exactement` rendent le protocole robuste aux lectures partielles de TCP.
2. Les **erreurs** sont typées : une erreur métier (« solde insuffisant ») est définitive ; une méthode inconnue est une erreur de protocole ; un **délai dépassé** donne un résultat **inconnu** (le serveur continue de travailler).
3. La **perte de réponse** reproduit l'incident de HashBox : le serveur a exécuté `debiter`, mais le client ne le sait pas ; sa relance (sémantique *au moins une fois*) débite **deux fois**. Le solde passe de 100 à 20 au lieu de 60.
4. Remède : rendre l'opération **idempotente** (mission 4).

À toi, avec aide

Un client lit des messages de ce flux d'octets TCP où les messages sont cadrés par un préfixe de longueur de 4 octets, **mais les données arrivent par morceaux de tailles quelconques** :
Écris un décodeur incrémental `Decodeur` avec une méthode `alimenter(octets)` qui renvoie la liste des messages complets disponibles (et conserve le reste). Teste en découpant un flux de 3 messages en morceaux de 1, 3, 7, puis 50 octets, dans cet ordre, et vérifie que tu retrouves exactement les 3 messages.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Garde un tampon `bytearray` entre les appels.

Indice 2

Boucle : si le tampon contient ≥ 4 octets et au moins 4 + taille octets, extrais un message.

Indice 3

Utilise `struct.unpack('>I', tampon[:4])`.

À toi, avec aide · réponse

Réponse

```
import json, struct

class Decodeur:
    def __init__(self): self.tampon = bytearray()
    def alimenter(self, octets):
        self.tampon += octets
        messages = []
        while len(self.tampon) >= 4:
            (taille,) = struct.unpack(">I", self.tampon[:4])
            if len(self.tampon) < 4 + taille: break
            messages.append(json.loads(bytes(self.tampon[4:4 + taille])))
            del self.tampon[:4 + taille]
        return messages

def encoder(objet):
    corps = json.dumps(objet).encode(); return struct.pack(">I", len(corps)) + cor
   ps

MSGS = [{"n": 1}, {"n": 2, "texte": "bonjour ça va ?"}, {"n": 3, "liste": list(ran
```

À toi, avec aide · réponse

Réponse (suite)

```
   ge(10))}]
flux = b"".join(encoder(m) for m in MSGS)
d, recus, i = Decodeur(), [], 0
for taille in (1, 3, 7, 50, 10_000):
    recus += d.alimenter(flux[i:i + taille]); i += taille
print(recus)
assert recus == MSGS and not d.tampon
```

Un message peut arriver **coupé** (début du préfixe, ou corps incomplet) ou **collé** à d'autres. Le décodeur incrémental est la forme robuste de ce que `lire_exactement` fait dans l'exemple. Tous les protocoles binaires sur TCP font cela.

À toi, en solo

Contrat évolutif. Écris un petit sérialiseur/désérialiseur de messages versionnés :

1. un schéma décrit des champs numérotés avec type et caractère obligatoire/optionnel (`{1: ("membre", str, True), 2: ("montant", int, True)}`) ;
2. `encoder(schema, message)` produit un dictionnaire `{numero: valeur}` (comme Protocol Buffers : le **numéro**, pas le nom, circule) ;
3. `decoder(schema, donnees)` : **ignore les champs inconnus**, applique des valeurs par défaut pour les champs optionnels absents, et lève une erreur claire pour un champ obligatoire absent ou mal typé ;
4. teste l'évolution : un serveur en version 2 (qui ajoute un champ optionnel `devise`) lit les messages d'un client version 1 ; un client version 1 lit les messages d'un serveur version 2 (champ inconnu ignoré) ;
5. montre un changement **cassant** : retirer un champ obligatoire, ou changer son type.

À toi, en solo · réponse

Pistes de correction

```
class ErreurContrat(Exception): pass

def encoder(schema, message):
    donnees = {}
    for numero, (nom, type_, obligatoire) in schema.items():
        if nom in message:
            if not isinstance(message[nom], type_): raise ErreurContrat(f"{nom} :
   type {type(message[nom]).__name__} au lieu de {type_.__name__}")
            donnees[numero] = message[nom]
        elif obligatoire:
            raise ErreurContrat(f"champ obligatoire absent : {nom}")
    return donnees

def decoder(schema, donnees, defauts=None):
    defauts, resultat = defauts or {}, {}
    for numero, (nom, type_, obligatoire) in schema.items():
        if numero in donnees:
            if not isinstance(donnees[numero], type_): raise ErreurContrat(f"{nom}
    : mauvais type")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
            resultat[nom] = donnees[numero]
        elif obligatoire:
            raise ErreurContrat(f"champ obligatoire absent : {nom}")
        elif nom in defauts:
            resultat[nom] = defauts[nom]
    return resultat                                          # les champs inconnus
   (numéros absents du schéma) sont ignorés

V1 = {1: ("membre", str, True), 2: ("montant", int, True)}
V2 = {**V1, 3: ("devise", str, False)}                       # ajout d'un champ OPT
   IONNEL : compatible
msg_v1 = encoder(V1, {"membre": "ada", "montant": 40})
msg_v2 = encoder(V2, {"membre": "ada", "montant": 40, "devise": "EUR"})
assert decoder(V2, msg_v1, defauts={"devise": "EUR"}) == {"membre": "ada", "montan
   t": 40, "devise": "EUR"}    # nouveau serveur, ancien client
assert decoder(V1, msg_v2) == {"membre": "ada", "montant": 40}
                                  # ancien serveur, nouveau client : champ ignoré
# changements cassants
V3_retire = {1: ("membre", str, True)}
```

À toi, en solo · réponse

Pistes de correction (suite)

```
                                  # retire montant
V3_type = {1: ("membre", str, True), 2: ("montant", str, True)}
                                  # change le type
try: decoder({**V1, 4: ("reference", str, True)}, msg_v1); raise AssertionError
                                  # nouveau champ OBLIGATOIRE
except ErreurContrat as e: print("cassant (champ obligatoire ajouté) :", e)
try: decoder(V3_type, msg_v1); raise AssertionError
except ErreurContrat as e: print("cassant (type modifié) :", e)
assert decoder(V3_retire, msg_v1) == {"membre": "ada"}         # un lecteur qui re
   tire un champ ne casse pas à la lecture, mais les consommateurs de « montant »
   perdent leur donnée
```

La règle de Protocol Buffers : **ne jamais réutiliser un numéro de champ**, ajouter plutôt que modifier, rendre les nouveaux champs optionnels. Ainsi, anciens et nouveaux programmes cohabitent pendant les déploiements progressifs.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Relancer un appel non idempotent après un délai	Doublons (débits, envois, créations)	Idempotence (mission 4) ; relances seulement si sûres
Appel sans délai d'attente	Un appelant bloqué indéfiniment	Échéance explicite sur chaque appel
Lire TCP comme s'il envoyait des messages	Messages tronqués ou collés	Cadrage et lecture complète
Contrat modifié de façon cassante	Clients anciens cassés pendant les déploiements	Champs optionnels, numéros stables, ignorer l'inconnu

Mini-boss

Mini-boss

Mini-framework RPC de HashBox. Construis `minirpc.py` :

1. cadrage par longueur, JSON, boucle de réception robuste ;
2. serveur multi-clients avec registre de méthodes, erreurs typées (métier, protocole, interne) ;
3. client avec souche dynamique, délai et **échéance** transmise dans la requête (le serveur ne démarre pas un travail dont l'échéance est dépassée) ;
4. identifiant de requête et **déduplication** côté serveur (cache des réponses pour les `id` déjà traités), pour obtenir « au plus une fois » apparent sur relance ;
5. injection de pannes (perte de réponse, délai, coupure de connexion) avec un `Chaos` configurable ;
6. 12 tests : succès, erreurs, délai, échéance, relance sans déduplication (double effet) et avec déduplication (effet unique), contrat compatible.

Mini-boss · réponse

Pistes de correction

Un bon travail : structures `Requete(id, methode, params, echeance)`, `Reponse(id, ok|erreur)`, cache des réponses `{id_client: {id: reponse}}` borné; un test central montre qu'avec une perte de réponse, le client qui relance avec le **même identifiant** obtient la réponse mémorisée sans ré-exécution. La conclusion attendue : le transport ne peut garantir « exactement une fois » ; c'est la combinaison identifiant stable + déduplication qui produit un **effet** unique.

Quiz éclair

Quiz éclair

1. Un délai dépassé sur un appel distant laisse le résultat :

A. forcément échoué
B. forcément réussi
C. inconnu : l'action a pu avoir lieu ou non

2. « Exactement une fois » s'obtient en pratique par :

A. au moins une fois (relances) + idempotence
B. une garantie du réseau
C. un délai plus long

3. Sur TCP, pour recevoir un message complet, il faut :

A. lire une seule fois

Quiz éclair

Quiz éclair (suite)

B. lire en boucle jusqu'à avoir tous les octets annoncés
C. fermer la connexion

4. 🔁 *Révision mission 01* : $a \rightarrow b$ (a s'est produit avant b) signifie :

A. que l'horloge physique de a est plus petite
B. que a et b sont sur la même machine
C. que a peut avoir influencé causalement b

5. 🔁 *Révision mission 01* : Deux événements aux horloges vectorielles [2,0] et [0,1] sont :

A. concurrents
B. l'un avant l'autre
C. identiques

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

1. Quel appel distant de ton système n'est pas sûr à rejouer, et que fais-tu aujourd'hui en cas de délai dépassé ?
2. Quel contrat de ton système est modifié sans que ses consommateurs soient prévenus ?

À retenir

À retenir

1. Un appel distant a trois issues : succès, échec, inconnu.
2. Cadrage par longueur et lecture complète sur TCP.
3. Au plus une fois, au moins une fois, exactement une fois = relances + idempotence.
4. Échéances explicites ; erreurs métier, transitoires, de protocole.
5. Contrats évolutifs : ajouter en optionnel, ne jamais réutiliser un numéro.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Messager fiable

Prochaine mission : Latence, timeout et retry

